EECS 280 – Lecture 23

Containers of Pointers and What's Next

6/15/2022

Review: Using Exceptions

- The **exception** mechanism introduces an additional **control flow path** for error handling.

```
int main() {
  int x = askUser();
  try {
    int f = factorial(x);
    if (f < 100) {
      cout << "Small" << endl;
    }
    else {
      cout << "Larger" << endl;
    }
  }
  catch (const FactorialError &e) {
    cout << "ERROR" << endl;
  }
}
```

Put code that might throw in a try block.

In separate code, we catch the exception and handle the error.

```
class FactorialError { };

// Returns n! for non-negative
// inputs. Throws an exception
// on negative inputs.
int factorial(int n) {

  // Check for error
  if (n < 0) {
    throw FactorialError();
  }
  ...
}
```

When something goes wrong, we throw an exception.

Error Handling vs. Undefined Behavior

- Error-handling mechanisms rely on well-defined behavior from functions that detect an error.

```
// EFFECTS:  Returns the price for the given item.
//           If the item doesn't exist, throws an
//           InvalidOrderException.
double getPrice(const string &item) const;
```

Error detection described in EFFECTS; no REQUIRES clause.

- Undefined behavior cannot generally be detected by the error mechanisms we saw.

```
try {
  int *total = nullptr;
  cout << *total << endl;
}
catch (...) {
  cout << "Undefined behavior" << endl;
}
```

But this will immediately crash

catch everything

Undefined behavior.

This does not run.

Try/catch does not handle a null-pointer dereference.

Review: List Template

List.h

```
template <typename T>
class List {
public:
  void push_front(T v);
  T & front();
private:
  struct Node {
    T datum;
    Node *next;
  };
  Node *first;
};
```

The compiler instantiates the template as needed according to how it is used in the code.

```
#include "List.h"
int main() {
  List<int> list1;
  List<Duck> list2;
}
```

```
class List<int> {
public:
  void push_front(int v);
  int & front();
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
};
```

```
class List<Duck> {
public:
  void push_front(Duck v);
  Duck & front();
private:
  struct Node {
    Duck datum;
    Node *next;
  };
  Node *first;
};
```

Exercise: Avoiding Copies

- Avoiding unnecessary copies can be crucial to writing efficient code!
- For example, consider a very large object like a `Gorilla`. It is very expensive to copy.
- Trace through this code…
  How many copies are made?

```
int main() {
  Gorilla g("Colo");
  List<Gorilla> zoo;
  zoo.push_front(g);

}
```

T = Gorilla

```
template <typename T>
void List<T>::push_front(T datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = first;
  first = p;
}
```

pass by value

pass by value

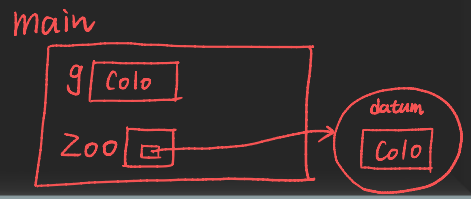
Colo was the name of the first Gorilla born in captivity.

Exercise: Avoiding Copies

- Avoiding unnecessary copies can be crucial to writing efficient code!
- For example, consider a very large object like a `Gorilla`. It is very expensive to copy.
- Trace through this code…
  How many copies are made?

T = Gorilla

```
int main() {
  Gorilla g("Colo");
  List<Gorilla> zoo;
  zoo.push_front(g);
}
```




```
template <typename T>
void List<T>::push_front(T datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = first;
  first = p;
}
```

Const T & datum

Solution: Avoiding Copies

- Two copies are made:
  - Passing the `datum` parameter by value
  - Assigning `datum` to `p->datum`
- We can avoid the first one if we pass the parameter by reference instead.
- If `T` is `int`, it's no big deal, but it could also be something huge like `Gorilla`!

2 Copies

```
template <typename T>
void List<T>::push_front(T datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = first;
  first = p;
}
```

1 Copy

```
template <typename T>
void List<T>::push_front(
  const T &datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = first;
  first = p;
}
```

The C++ Zoo

- Francine the zookeeper works at the C++ Zoo.
- She makes a list of the llamas she needs to feed.

```
int main() {
  Llama l1("Paul");
  Llama l2("Carl");

  List<Llama> todo;
  todo.push_back(l1);
  todo.push_back(l2);

  for (auto &llama : todo) {
    // feed each Llama in the list
  }
}
```

We make a copy of llamas in our list, so we feed the clones of these llamas. And never feed the original one.

What's wrong with this code?

Containers and Value Semantics

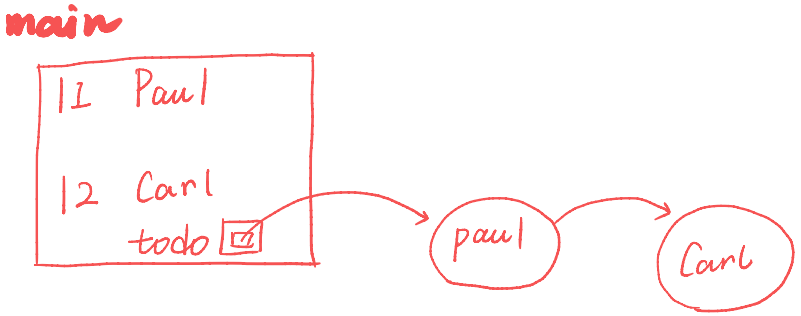
- When we add an object to a container, what we're really doing is **adding a copy** of it!

```
int main() {
  Llama l1("Paul");
  Llama l2("Carl");

  List<Llama> todo;
  todo.push_back(l1);
  todo.push_back(l2);

  for (auto &llama :
       todo) {
    // feed each Llama
    // in the list
  }
}
```

Exercise: Draw out Memory



Containers of Pointers

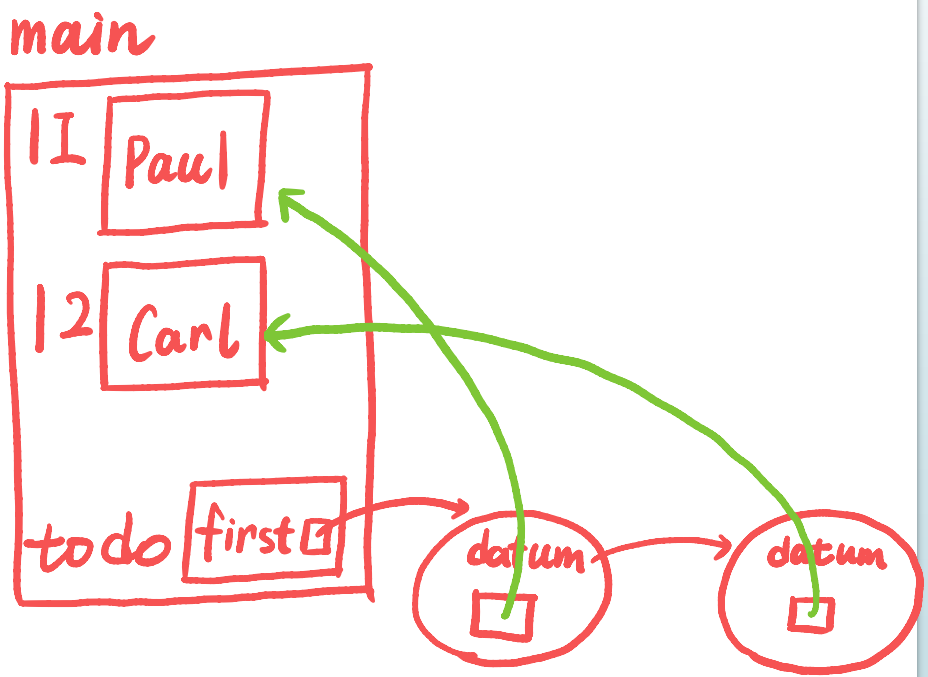
- Use a **container of pointers** instead. The only thing copied is the address, not the original!

```
int main() {
  Llama l1("Paul");
  Llama l2("Carl");

  List<Llama*> todo;
  todo.push_back(&l1);
  todo.push_back(&l2);

  for (auto lptr : todo) {
    // dereference each
    // Llama ptr in the list
    // and feed the llama
  }
}
```

Exercise: Draw out Memory

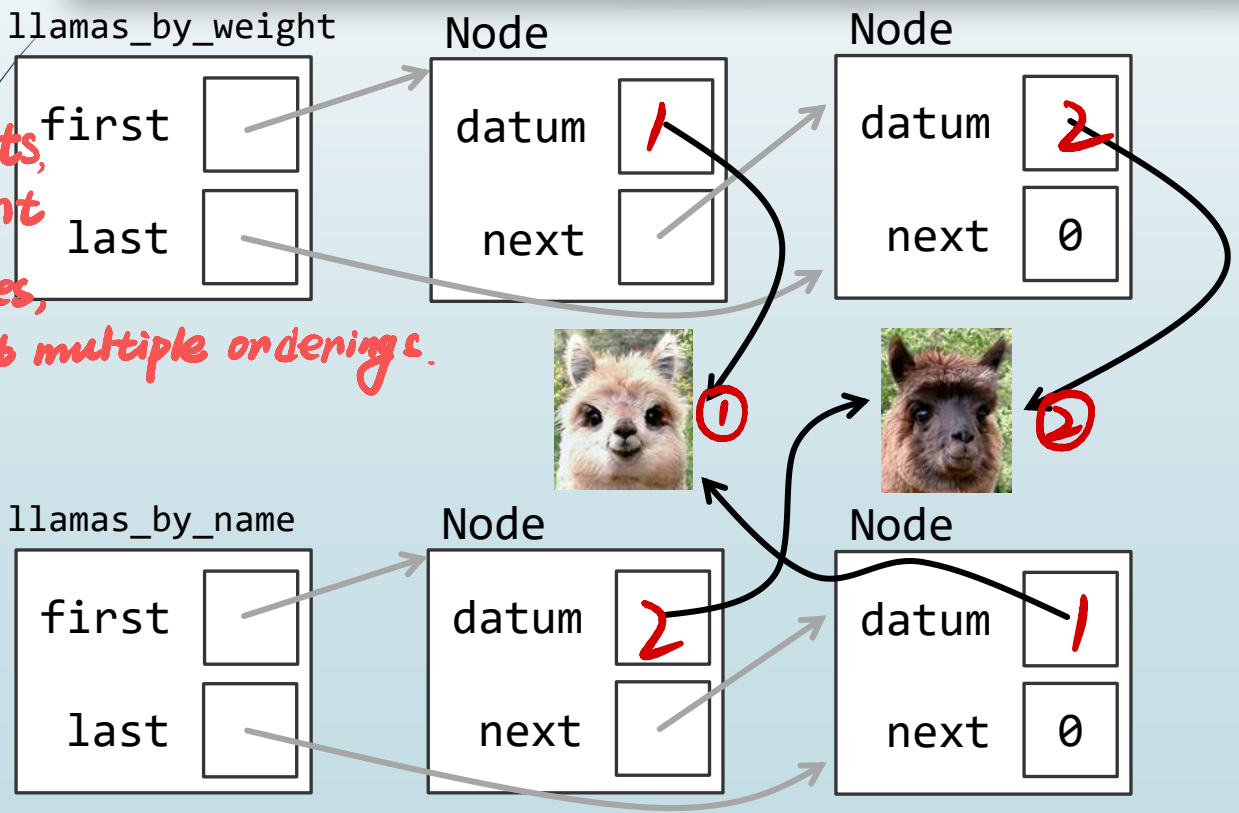


Containers of Pointers

- Use containers of pointers to work with two different orderings of one set of objects.

```
List<Llama*> llamas_by_weight;
List<Llama*> llamas_by_name;
```

if you got a set of elements, you don't want to make copies, and you want multiple orderings.



Containers of Dynamic Objects

- You can also use containers of pointers to keep track of dynamically allocated objects.
- As always, you must remember to clean up the dynamic memory. What's wrong with this code?

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_back(new Gorilla("Colo"));
  zoo.push_back(new Gorilla("Koko"));
  // do something with Gorillas
  // main ends
}
```

for the destructor of list does not clean up these gorillas for us. it will clean up the dynamic memory for the nodes. But the nodes contain pointers to other dynamic memory, the linked list doesn't know about that.

Orphaned Gorilla on the heap!

这是一件好事，因为如果 list 的 destructor 可以沿着 pointer 自动 delete 掉 Gorilla 的话，在之前的例子中，在 stack 中的两个 Gorilla 就会被错误 delete（因其不在 heap）

Containers of Dynamic Objects

- You can also use containers of pointers to keep track of dynamically allocated objects.
- As always, you must remember to clean up the dynamic memory.

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_back(new Gorilla("Colo"));
  zoo.push_back(new Gorilla("Koko"));

  // do something with Gorillas
  while (!zoo.empty()) {
    delete zoo.front();
    zoo.pop_front();
  }
}
```

← its our job instead of list job to clean them.

I'm free!

Containers of Dynamic Objects

- Only one container should "own" the dynamic object and is responsible for cleaning it up.

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_back(new Gorilla("Colo"));
  zoo.push_back(new Gorilla("Koko"));
  List<Gorilla*> todo = zoo;

  // do something with Gorillas

  while (!zoo.empty()) {
    delete zoo.front();
    zoo.pop_front();
  }

  while (!todo.empty()) {
    delete todo.front();
    todo.pop_front();
  }
}
```

we only need to delete through one of the lists. And then the other list would be a list of dangling pointers. just make sure not use it

A double free occurs for each Gorilla.

Keeping Track of Dynamic Objects

- Potential pitfalls with containers of pointers to dynamically allocated objects include:
  - Using an object after it's been deleted.
  - Leaving an object orphaned (forgetting to delete).
  - Accidentally deleting an object twice.

- To fix these bugs, make sure that every dynamic object has a single "owner".
  - If multiple containers have pointers to a single dynamic object, exactly one of them should be "in charge" of it.
  - If the pointer is removed, the object should either be deleted or ownership should be transferred elsewhere

Sorting Containers of Pointers

- What does this code do?

```
int main() {
  vector<Gorilla*> zoo;
  zoo.push_back(new Gorilla("Colo"));
  zoo.push_back(new Gorilla("Koko"));

  std::sort(zoo.begin(), zoo.end());
}
```

- It sorts the gorillas based on their addresses! This will compile, but it's probably not what we wanted!

Sorting Containers of Pointers

- You'll need to create a comparator that dereferences the pointers, and then provide that to `std::sort`.
- For example:

```
class GorillaNameLess_ptr {
public:
  bool operator()(const Gorilla *g1, const Gorilla *g2) const {
    return g1->getName() < g2->getName();
  }
};

int main() {
  vector<Gorilla*> zoo;
  zoo.push_back(new Gorilla("Colo"));
  zoo.push_back(new Gorilla("Koko"));

  std::sort(zoo.begin(), zoo.end(), GorillaNameLess_ptr());
}
```

Keeping Multiple Orderings

- With containers of pointers, you can maintain several different sorted orderings of the same objects.

```
int main() {
  vector<Gorilla*> zoo;
  zoo.push_back(new Gorilla("Colo"));
  zoo.push_back(new Gorilla("Koko"));

  vector<Gorilla*> byName = zoo;
  std::sort(byName.begin(), byName.end(),
            GorillaNameLess_ptr());

  vector<Gorilla*> byWeight = zoo;
  std::sort(byWeight.begin(), byWeight.end(),
            GorillaWeightLess_ptr());

}
```

Non-Dynamic Objects

- Containers of pointers don't have to work with dynamic memory. Here's an example:

```
int main() {
  vector<Gorilla> gorillas;
  gorillas.push_back(Gorilla("Colo"));
  gorillas.push_back(Gorilla("Koko"));
  ...

  // an alternate ordering, using pointers to originals
  vector<Gorilla*> byName;
  for (auto &g : gorillas) {
    byName.push_back(&g);
  }

  GorillaNameLess_ptr gpnc;
  std::sort(byName.begin(), byName.end(), gpnc);

}
```

The actual Gorillas live in the vector.

Pointers to the original gorillas.

A new ordering for the pointers, but not the originals.

A special comparator that operates on Gorilla*.

Containers of Polymorphic Objects

- A container can only contain one kind of element.
- However, we can effectively have a container of many different derived types through polymorphism!

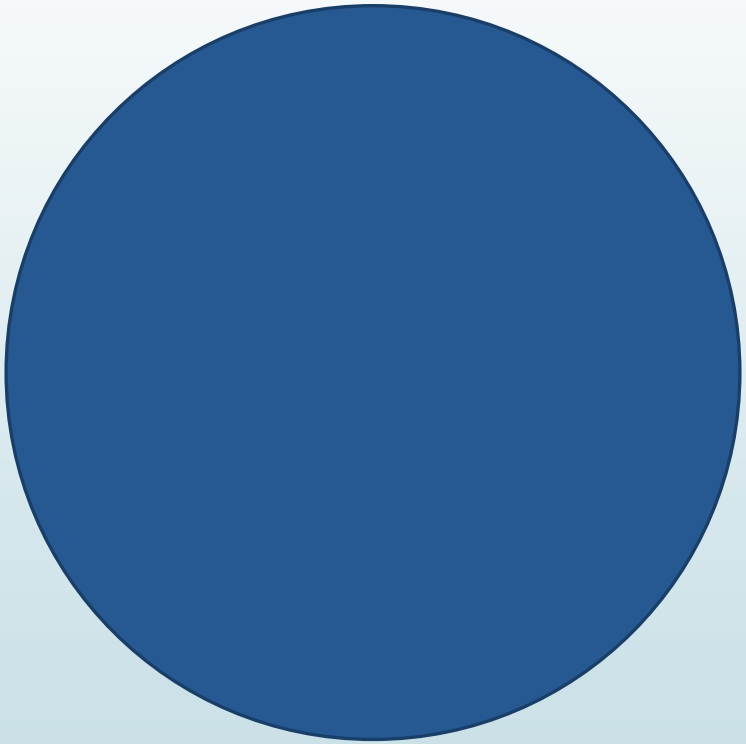
```
int main() {
  vector<Animal*> zoo;
  zoo.push_back(new Gorilla("Colo"));
  zoo.push_back(new Llama("Susie"));
  zoo.push_back(new Unicorn("Charlie"));
  zoo.push_back(new Rabbit("Judy"));

  for (auto animal_ptr : zoo) {
    animal_ptr->talk(); // a virtual function
  }

  // prints different messages for each animal
}
```

Break Time!

We'll start again in five minutes.



What EECS 280 is about…

- Generalizable CS concepts
  - Procedural Abstraction
  - Data Abstraction
  - Dynamic Resource Management
  - And much more!
- Building programming skills
  - Learn conceptually "what code does"
  - Implement large programming projects

CS-Related Programs

- CS-LSA: http://cs.lsa.umich.edu/undergraduate-cs-programs/
- CS-Eng: https://www.eecs.umich.edu/eecs/undergraduate/computer-science/
- CS minor: https://www.eecs.umich.edu/eecs/undergraduate/cs-minor/
- Computer Engineering (CE): https://www.eecs.umich.edu/eecs/undergraduate/ugce/computer_engineering.html
- Electrical Engineering (EE): https://www.eecs.umich.edu/eecs/undergraduate/ugee/electrical_engineering.html
- Data Science (DS-LSA): http://lsa.umich.edu/stats/undergraduate-students/undergraduate-programs/majordatascience.html
- Data Science (DS-Eng): https://www.eecs.umich.edu/eecs/undergraduate/data-science/

Declaring the CS-LSA Major

- Pre-declaration courses
  - Math 115
  - Math 116
  - EECS 203 (or equivalent)
  - EECS 280
- Must obtain at least a C in each course.
- Some courses may be transferred (though it's rare for external courses to transfer as EECS 280.)
- Talk to an advisor if you have questions or concerns about your situation.

The other programs do not require 280 to declare, so we won't discuss their requirements. Please see an advisor for the respective program if you have questions.

CS Major Requirements

- Common requirements for CS-LSA and CS-Eng:
  - EECS 281
  - EECS 370
  - EECS 376
  - Stats 250 (or equivalent)
  - 4 Upper Level CS (ULCS) electives
  - Capstone/Major Design Experience (MDE)

- See the program guides for additional requirements or requirements for other majors.

CS-LSA also has optional tracks that can take the place of the 4 ULCS courses. See the program guide for details.

CS Minor Requirements

- Core courses:
  - EECS 203
  - EECS 280
  - EECS 281
- One CS minor elective course
  - Currently one of EECS 388, 482, 483, 484, 485, 487, 490, 492, or 493
- Optional (not required): EECS 370 and 376

Follow-on Courses

- EECS 281: Data Structures and Algorithms
  - An introduction to algorithm analysis and more advanced data structures and programming techniques.
- EECS 370: Introduction to Computer Organization
  - Learn about the basic concepts of computer organization, hardware, and how computers execute programs.
- EECS 376: Foundations of Computer Science
  - An introduction to the theory of computation, including models of computation, computability, and complexity.
- (Optional) EECS 201: Computer Science Pragmatics
  - Learn tools such as shells, scripting, Makefiles, version control, text editors, and debugging.